Streaming Partitioning of Sequences and Trees*

Christian Konrad

School of Computer Science, Reykjavík University, Reykjavík, Iceland
christiank@ru.is

Abstract

We study streaming algorithms for partitioning integer sequences and trees. In the case of trees, we suppose that the input tree is provided by a stream consisting of a depth-first-traversal of the input tree. This captures the problem of partitioning XML streams, among other problems.

We show that both problems admit deterministic $(1+\epsilon)$-approximation streaming algorithms, where a single pass is sufficient for integer sequences and two passes are required for trees. The space complexity for partitioning integer sequences is $O(\frac{1}{\epsilon} p \log(nm))$ and for partitioning trees is $O(\frac{1}{\epsilon} p^2 \log(nm))$, where n is the length of the input stream, m is the maximal weight of an element in the stream, and p is the number of partitions to be created.

Furthermore, for the problem of partitioning integer sequences, we show that computing an optimal solution in one pass requires $\Omega(n)$ space, and computing a $(1+\epsilon)$-approximation in one pass requires $\Omega(\frac{1}{\epsilon} \log n)$ space, rendering our algorithm tight for instances with $p, m \in O(1)$.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Streaming Algorithms, XML Documents, Data Partitioning, Communication Complexity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2016.13

1 Introduction

Partitioning Massive Data Sets. *Data partitioning* is a widely employed technique for processing massive data sets. The input data is partitioned into (not necessarily disjoint) subsets of much smaller sizes which are then distributed to different computational units. Parallel or distributed algorithms are then executed on the partitioned data.

The data partitioning step can be a difficult task in itself, especially if the data sets considered are massive. Some partitioning problems are NP-hard (some are even hard to approximate [3]), while others are more amenable. However, in the context of massive data sets, it is not clear whether even the more amenable problems can be solved efficiently.

In this paper, we are therefore interested in how well big data sets can be partitioned by massive data set algorithms. We focus on streaming algorithms, and we consider the problems of partitioning integer sequences and partitioning trees. Streaming algorithms use a small random access memory which is usually only of poly-logarithmic size in the input. They scan the entire data from left to right sequentially in passes and therefore make optimal use of data locality.

Partitioning Integer Sequences. Let $X \in \{0, \ldots, m\}^n$ be a sequence of integers of length n, for an integer m. Given an integer parameter p, the goal is to partition X into p contiguous blocks so as to minimize the maximum weight (sum of elements) of a block. In other words, we have to find $p-1$ separators $s_1, \ldots, s_{p-1}$ with $1 = s_0 \leq s_1 \leq \cdots \leq s_{p-1} \leq s_p = n+1$ such

* Supported by Icelandic Research Fund grants 120032011 and 152679-051.

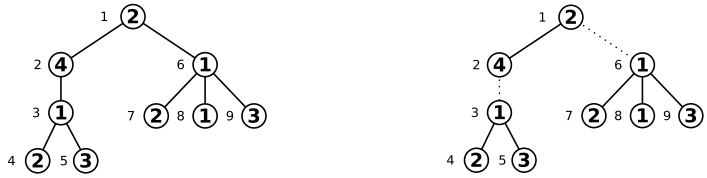

Figure 1 *Left*: A weighted tree t. The small numbers next to the nodes denote their IDs. The trace of a depth-first-traversal of t is $241\overline{2}\overline{3}3\overline{1}\overline{4}12\overline{2}\overline{1}\overline{1}3\overline{3}\overline{1}\overline{2}$, where we ambiguously write w for ('d', w) and $\overline{w}$ for ('u', w). *Right*: Partitioning of t into three parts with bottleneck value 7. A streaming algorithm should output the IDs of the root nodes of the created partitions. Here, these are $1, 3, 6$.

that $\max\left\{\sum_{i=s_j}^{s_{j+1}-1} X_i \;\middle|\; j \in \{0,\ldots,p-1\}\right\}$ is minimized. We will abbreviate this problem as Part.

This partitioning problem appears in many applications, especially in the context of load balancing, and has been extensively studied both from a theoretical [6, 9, 11, 20, 21, 13, 10] and a practical perspective [22, 25]. One example application is the decomposition of large computational meshes along space-filling curves [26, 15, 4] in parallel scientific computing, where multi-dimensional grid elements are ordered with respect to a traversal along a space-filling curve, giving rise to the one-dimensional problem of partitioning integer sequences. Very efficient exact algorithms for this problem exist, for example the $O(n \log n)$ time algorithm of Khanna et al. [13], the $O(n + p^{1+\epsilon})$ time algorithm of Han et al. [10], and a highly non-trivial optimal $O(n)$ time algorithm of Frederickson [9].

Partitioning Trees. The problem of partitioning integer sequences is extended to trees as follows: Given a rooted unranked node-weighted tree t with weights taken from the set $\mathcal{U} = \{0,\ldots,m\}$ for an integer m, and an integer p, the goal is to partition t into p subtrees $t_1,\ldots,t_p$ by removing $p-1$ edges so as to minimize the maximum weight of a subtree. If we are allowed random access to the input, this problem can also be solved optimally by an $O(n)$ time algorithm by Frederickson [9]. Classical applications of this problem include paging and overlaying [24]. More importantly, this problem also captures the problem of partitioning XML documents (see further below), which is the main motivation of this work.

In the case of integer sequences, we assume that the input stream for our streaming algorithm is the integer sequence X itself. In the case of trees, since we target XML documents, we assume that the input stream constitutes the trace of a depth-first-traversal of the input tree t. More precisely, the input stream is the following trace $X \in (\{\text{'d'},\text{'u'}\} \times \mathcal{U})^{2n}$ of a depth-first-traversal of t ('d' stands for down-step and 'u' stands for up-step): If a node with weight $w \in \mathcal{U}$ is visited top-down (bottom-up) at step i then $X_i = (\text{'d'}, w)$ (resp. $X_i = (\text{'u'}, w)$). We also say that a node v of the input tree t has ID i if it is the ith node that is visited by the depth-first-traversal. The trace X is fed into our streaming algorithm, and we require that the algorithm outputs the IDs of the root nodes of the p partitions (or subtrees). We will denote this problem by Tree. In Figure 1, we give an example tree illustrating the trace of a depth-first-traversal as well as a partitioning of the example tree.

Partitioning XML Documents and Other Applications. The traces of depth-first-traversals of trees constitute a Dyck language, and, vice versa, every word of a Dyck language can be seen as the concatenation of traces obtained from depth-first-traversals of the trees of a forest. Dyck languages are languages of well-parenthesized expressions, and algorithms that process Dyck languages such as [29, 19] and our work therefore have potential applications

in all areas where those types of expressions appear. These include arithmetic expressions, the sequence of `CALL` and `RETURN` instructions of the traces of program executions, and XML documents, which are probably the most relevant application for the database community. An XML document can be seen as the trace of a depth-first-traversal of its underlying document tree.

Partitioning XML documents is for instance widely used in the area of parallel XML databases and for the parallel evaluation of queries such as XPath, where the term *XML fragmentation* is usually employed [7]. In this context, additional constraints on the partitioning other than load balancing are often imposed in order to avoid data dependencies between different partitions, however, achieving good load balancing is crucial for any performant solution. Kim and Kang [14] study the setting where an XML document is fragmented and the resulting partitions are streamed to mobile client devices, which then distributively evaluate a query. They underline the importance of small partitions by pointing out that: "No matter how efficient a fragmentation method is, it is useless and impractical unless the size of every resulting fragment is less than that of the buffer allocated for receiving the fragments at the client devices." Thus, our work can be seen as a first step towards streaming XML fragmentation. Further potential applications of the tree partitioning problem include the parallel validation of XML documents with respect to local validity schemata such as DTDs [16] and similar problems that require only local knowledge of an XML document.

Starting Point: Parametric Search. Many previous works tackle Part and Tree using *parametric search* [9, 10, 13]. For both problems, given a value B, there is an algorithm that computes a partitioning in linear time with bottleneck value at most B if such a partitioning exists. If there is no such partitioning, then the algorithm fails. Such an algorithm can be regarded as a feasibility tester: Given a value B, it checks whether the bottleneck value of an optimal partitioning B^* is larger than B (B is not feasible) or smaller than/equal to B (B is feasible). A trivial range for B^* is $\{0, 1, \ldots, mn\}$[1] since the input consists of n integers or tree nodes of maximal weight m. Thus, via a binary search, running the feasibility tester $\mathrm{O}(\log(mn))$ times, an exact algorithm with runtime $\mathrm{O}(n \log(mn))$ can be obtained.

For an integer sequence X, a value B can be tested by traversing X from left to right, setting up partitions of maximal sizes at most B. If at most p partitions are created, then B passes the test, otherwise it fails. This tester, denoted Probe in the literature, can be implemented as a one-pass streaming algorithm with $\mathrm{O}(p \log(n) + \log(mn))$ space[2]. Hence, using the previous binary search, a $\mathrm{O}(\log(mn))$ passes, $\mathrm{O}(p \log(n) + \log(mn))$ space exact streaming algorithm for Part can be obtained. For Tree, a linear time feasibility tester also exists [9], however, it appears impossible to implement it space efficiently as a one-pass streaming algorithm.

Probe can also be used to obtain a one-pass streaming algorithm with approximation factor $(1+\epsilon)$ (meaning that a partitioning with bottleneck value at most by a factor $(1+\epsilon)$ larger than the optimal bottleneck value is computed): In one pass, we run multiple copies of Probe testing values $(1+\epsilon)^i$, for all integers i with $0 \leq i \leq \frac{\log(mn)}{\log(1+\epsilon)} = \mathrm{O}(\frac{\log(mn)}{\epsilon})$, in parallel, and we return the partitioning with the smallest feasible bottleneck value. We observe that only $\mathrm{O}(\frac{\log p}{\epsilon})$ copies of Probe are active at any moment during the processing of the stream:

[1] A slightly better upper bound for B^* is $\lceil \frac{mn}{p} \rceil + m$, however, this does not change our arguments and only complicates the presentation.

[2] $\mathrm{O}(p \log(n))$ for storing $p-1$ partition boundaries, and $\mathrm{O}(\log(mn))$ for accumulating the weight of the current partition.

Let B' be the smallest bottleneck value that has not yet been declared infeasible by a copy of Probe. Then, the total weight of the prefix of the stream seen so far is at most pB', and hence the copies of Probe testing values larger than pB' have not yet set their first partition boundaries. Thus, it is not necessary to explicitly execute them yet. Using this observation, the following result can be obtained:

▶ **Fact 1.** *There is a one-pass* $(1+\epsilon)$*-approximation streaming algorithm for* Part *with space* $O(\frac{\log p}{\epsilon} \cdot (p \log(n) + \log(mn)))$.

We regard this algorithm as a baseline against which we will compare our results. Its main weakness is its update time [3]: Since $\Theta(\frac{\log p}{\epsilon})$ copies of Probe have to be updated, the worst-case and amortized update time is $\Theta(\frac{\log p}{\epsilon})$. Furthermore, the $\log p$ factor in the space complexity of the algorithm seems unnatural. In this paper, we will show that, using a very different technique, both weaknesses can be overcome.

Results and Techniques. In many areas of computer science, a common technique for dealing with large objects is to replace them with smaller ones that capture important properties of the initial object sufficiently well. Examples include kernelization [18], approximate distance oracles [27, 23], and graph sparsification [5]. In recent years, this technique has proved useful for data streaming algorithms, e.g., [8, 1, 2, 12], and we follow this line here.

Our algorithms for Part and Tree compute much smaller instances from the problem instance described in the input stream, so that a $(1+\epsilon)$-approximate partitioning can be deduced from an exact partitioning of the small instances. Our contribution is two-fold: First, we identify the right properties that guarantee that the smaller objects still capture $(1+\epsilon)$-approximate partitionings of the original instance. In the case of Part, we prove that a small instance of size $O(\frac{p}{\epsilon})$ is sufficient, and for Tree, an instance of size $O(\frac{p^2}{\epsilon})$ suffices. Note that, in both cases, the size is independent of n. Second, we prove that the small objects with the right properties can be computed space efficiently in the streaming model. Then, in a post-processing step, we use exact algorithms for partitioning the small instances.

This technique leads to the following algorithmic results:

- A deterministic one-pass $(1+\epsilon)$-approximation streaming algorithm for Part with space $O(p \log(mn)/\epsilon)$, worst case update time $O(1)$, and post-processing time $O(p/\epsilon)$ (**Theorem 7**).
- A deterministic two-pass $(1+\epsilon)$-approximation streaming algorithm for Tree with space $O(p^2 \log(mn)/\epsilon)$, worst case update time $O(1)$ and post-processing time $O(p^2/\epsilon)$ (**Theorem 15**).

Note that our algorithm for Part improves on the space complexity and the update time of the algorithm described in Fact 1. Last, we complement our algorithms with lower bounds for Part obtained through results in communication complexity:

- Via a reduction to the one-way two-party communication problem Index, we show that computing an exact solution for Part in one pass requires $\Omega(n)$ space (**Theorem 18**).
- Via a more involved combinatorial argument, we show that algorithms that compute a $(1+\epsilon)$-approximation for Part require space $\Omega(\log(n)/\epsilon)$ (**Theorem 20**).

The latter lower bound shows that our algorithm for Part is best possible for $p, m \in O(1)$, and, in particular, that the $\frac{1}{\epsilon}$ factor in the space complexity is unavoidable for one-pass algorithms.

[3] The time between two consecutive read operations on the stream

Outline. We consider PART in Sec. 3 and TREE in Sec. 4. Our lower bounds are given in Sec. 5, and we conclude in Sec. 6. Due to space restrictions, the proofs of lemmas and theorems marked by **(*)** are omitted.

2 Notations and Definitions

Streaming Algorithms. Generally, we denote the input stream for our streaming algorithms by $X = X[1], X[2], \ldots$. In the case of integer sequences, the length of X is n, and in the case of trees, the length is $2n$, since each of the n nodes of the input tree is visited twice by a depth-first-traversal. A *streaming algorithm* that processes X reads the elements of X sequentially one-by-one in passes. Streaming algorithms are defined as follows:

▶ **Definition 1** (Streaming algorithm)**.** An algorithm **A** is a $p(n)$-pass deterministic/randomized *streaming algorithm* with memory $s(n)$, update time $t(n)$, amortized update time $a(n)$, and post-processing time $t'(n)$, if for every input stream X of length n:

1. **A** performs at most $p(n)$ passes over the input stream X,
2. **A** maintains a random access memory of size $s(n)$,
3. **A** has worst case running time $t(n)$ between two consecutive read operations,
4. **A** has average running time $a(n)$ between two consecutive read operations,
5. **A** has running time $t'(n)$ between the last read operation and the output of the result.

If **A** is randomized then **A** has access to an infinite number of independent random coin flips, and it outputs a correct solution with probability at least $2/3$.

The algorithms presented in this paper are deterministic, however, we include randomized algorithms in Definition 1 since our lower bounds also hold for randomized algorithms. Furthermore, we suppose that reading an element from the input stream takes $O(1)$ time.

Notations for Sequences. For an array X, we denote the ith element by either $X[i]$ or X_i. For $i \leq j$, the array consisting of $X[i], X[i+1], \ldots, X[j]$ is denoted by $X[i, j]$.

Notations for Trees. Let t be an unranked rooted tree on n nodes. We denote by $\mathrm{rt}(t)$ the root of t. For any node $v \in t$, we denote by $\mathrm{stree}_t(v)$ the subtree of t rooted at v. For two nodes $x, y \in t$, let $\mathrm{lca}_t(x, y)$ denote the lowest common ancestor of x and y in t, and for a subset of nodes $U \subseteq t$, let $\mathrm{lca}_t(U)$ denote the lowest common ancestor of all nodes of U in t (if $U = \{u\}$ then $\mathrm{lca}_t(U) = u$). Given a node $v \in t$, we denote its ID, i.e., the position of v with respect to a depth-first-traversal of t, by $\mathrm{Id}_t(v)$. Given an ID i, we denote its corresponding node in t by $\mathrm{node}_t(i)$. With this notation, we have $\mathrm{node}_t(\mathrm{Id}_t(v)) = v$.

3 Algorithm For Partitioning Sequences

The main idea of our algorithm is the computation of a *coarse version* Y of the input stream X. Intuitively, a coarse version is obtained from X by repeatedly merging subsequent integers into a single element whose weight is the sum of the merged elements. If the coarse version Y fulfills certain properties, it can be shown that, from an optimal partitioning of Y, a $(1+\epsilon)$-approximation to the optimal partitioning of X can be obtained.

In Subsec. 3.1, we define coarse versions. Then, in Subsec. 3.2, we show how an appropriate coarse version Y can be computed in one pass. In a post-processing step, the coarse version Y is partitioned optimally, giving a $(1+\epsilon)$-approximation to the optimal partitioning of X.

3.1 Coarse Version

We now define a c-coarse version of X, and we prove that an optimal partitioning of Y provides an approximate partitioning of X.

▶ **Definition 2** (c-coarse version)**.** Let $m, m', n, n' \in \mathbb{N}$. Let $X \in \{0, \ldots, m\}^n$ and $Y \in \{0, \ldots, m'\}^{n'}$ be integer sequences. Then Y is a *c-coarse version* of X if $n' \leq n$ and there is a mapping $f : \{1, \ldots, n\} \to \{1, \ldots, n'\}$, denoted the *coarsening function*, such that:

1. f is surjective and increasing,
2. For every $1 \leq i' \leq n' : \sum_{i \in f^{-1}(i')} X[i] = Y[i']$,
3. For every $1 \leq i' \leq n' : Y[i'] \leq X[\min f^{-1}(i')] + c$.

Suppose that $X = 1\,3\,2\,1\,1\,5$. Then, $4\,4\,5$ is a 3-coarse version of X where the grouping of the elements of X has been done as follows: $\mathbf{1}\,3 \,|\, \mathbf{2}\,1\,1 \,|\, \mathbf{5}$. In order to fulfill Item 3 of the previous definition, the bold elements of the previous grouping have to be mapped to elements in the 3-coarse version that are larger by at most 3.

Lemma 3 shows that an optimal partitioning of a c-coarse version has a bottleneck value that is at most by the additive term c larger than the bottleneck value of an optimal partitioning of the initial sequence.

▶ **Lemma 3.** *Let $m, m', n, n' \in \mathbb{N}$. Let $X \in \{0, \ldots, m\}^n$ be an integer sequence and let $Y \in \{0, \ldots, m'\}^{n'}$ be a c-coarse version of X with coarsening function f, for a parameter c. Let B^* be the bottleneck value of an optimal partitioning of X into p parts. Let $s'_0, s'_1, \ldots, s'_p$ be the separators of an optimal partitioning of Y into p parts and let B'^* be the bottleneck value. Then:*

1. *The separators $s_0, s_1, \ldots, s_{p-1}, n+1$ with $s_i = \min f^{-1}(s'_i)$ for $i = 0, \ldots, p-1$ induce a partitioning of X with bottleneck value B'^*,*
2. *$B'^* \leq B^* + c$.*

Proof. Concerning Item 1, it follows from Item 2 of Definition 2 that the weight of the partition induced by separators s'_i and s'_{i+1} in Y equals the weight of the partition induced by separators s_i and s_{i+1} in X, for every i. Therefore, the bottleneck value is also the same.

Concerning Item 2, consider an optimal partitioning of X into p parts with bottleneck value B^*, and let $s^*_0, \ldots, s^*_p$ denote the separators of this partitioning. We will argue that, given the s^*_i, we can compute a partitioning of Y with separators $\tilde{s}_i$ with bottleneck value at most $B^* + c$. Since the optimal partitioning of Y with separators $s'_0, s'_1, \ldots, s'_p$ is at least as good, the result follows.

We define $\tilde{s}_p = n' + 1$, and for $i = 0, \ldots p-1$, let $\tilde{s}_i = \min\{j : \min f^{-1}(j) \geq s^*_i\}$, i.e., partition i in Y starts with the first element whose pre-image starts in partition i in X.

Now we argue that for any i, the weight of partition i in Y is at most the weight of partition i in X plus c. This then proves Item 2, as this property also holds for the bottleneck partition. We have:

$$\sum_{j=\tilde{s}_i}^{\tilde{s}_{i+1}-1} Y[j] = \left(\sum_{j=\tilde{s}_i}^{\tilde{s}_{i+1}-2} Y[j]\right) + Y[\tilde{s}_{i+1}-1] \leq \left(\sum_{j=\min f^{-1}(\tilde{s}_i)}^{\max f^{-1}(\tilde{s}_{i+1}-2)} X[j]\right) +$$

$$+ \big(X[\min f^{-1}(\tilde{s}_{i+1}-1)] + c\big) = \left(\sum_{j=\min f^{-1}(\tilde{s}_i)}^{\min f^{-1}(\tilde{s}_{i+1}-1)} X[j]\right) + c \leq \left(\sum_{j=s^*_i}^{s^*_{i+1}-1} X[j]\right) + c.$$

For the first inequality, we used Item 2 of Definition 2 to rewrite the sum, and Item 3 of Definition 2 in order to bound $Y[\tilde{s}_{i+1}-1]$. For the second inequality, we extended the

Algorithm 1 Computation of coarse version

Require: Number of partitions p, parameter ϵ for $(1+\epsilon)$-approximation

$s \leftarrow 4\lceil \frac{p}{\epsilon} \rceil$ {Size of array Y}
$Y \leftarrow (Y_{i1}, Y_{i2})_i$ array of integer tuples of length s, initially all tuples are $(0,0)$
while input stream not empty **do**
  $k \leftarrow (\arg\min_{1 \le i \le s} \{Y_{i1} = 0\}) - 1$
  $x_1 \dots x_{s-k} \leftarrow$ next $s-k$ integers from input stream, if fewer than $s-k$ integers are left then interpret the missing ones as 0s
  $\forall i \in \{1, \dots, s-k\} : Y[k+i] \leftarrow (x_i, 0)$ {Append the x_i to Y}
  $S \leftarrow \sum_{i=1}^{s} Y_{i1} + Y_{i2}$ {Weight of Y, equals weight of input stream seen so far}
  $Y \leftarrow \text{CPROBE}(Y, \lfloor S\epsilon/p \rfloor)$
end while
$Y' \leftarrow$ integer array of length s with $\forall 1 \le i \le s : Y'[i] = Y_{i1} + Y_{i2}$
return Y'

Algorithm 2 CPROBE(Y, c)

Require: $Y = (Y_{i1}, Y_{i2})_i$ array of integer tuples of length s, integer c

$Z \leftarrow (Z_{i1}, Z_{i2})_i$ array of integer tuples of length s
$j \leftarrow 1$ {current element in Z}
$Z_j \leftarrow Y_1$
for $i = 2 \dots s$ **do**
  if $Z_{j2} + Y_{i1} + Y_{i2} \le c$ **then**
    $Z_{j2} \leftarrow Z_{j2} + Y_{i1} + Y_{i2}$
  else
    $j \leftarrow j+1$, $Z_j \leftarrow Y_i$
  end if
end for
return Z

sum using the observations $s_i^* \le \min f^{-1}(\tilde{s}_i)$, and $s_{i+1}^* - 1 \ge \min f^{-1}(\tilde{s}_{i+1} - 1)$. These observations follow from the definition of $\tilde{s}_i$. ◀

We conclude that in order to obtain a $(1+\epsilon)$-approximation, a $(S\epsilon/p)$-coarse version of X with $S = \sum_{i=1}^{n} X[i]$ is required.

▶ **Corollary 4.** *Let Y be a $(S\epsilon/p)$-coarse version of X where $S = \sum_{i=1}^{n} X[i]$, for some $\epsilon > 0$. An optimal partitioning of Y allows us to obtain a $(1+\epsilon)$-approximation to* PART *on X.*

Proof. Let B^* be the bottleneck value of an optimal partitioning of X. Clearly, $B^* \ge S/p$. By Lemma 3, we obtain the approximation factor: $\frac{B^* + S\epsilon/p}{B^*} = 1 + \frac{S\epsilon/p}{B^*} \le 1 + \frac{S\epsilon/p}{S/p} = 1+\epsilon$. ◀

3.2 Computing Coarse Versions

Algorithm. Our algorithm for computing a $(S\epsilon/p)$-coarse version of the input stream X with $S = \sum_{i=1}^{n} X[i]$ is depicted in Algorithm 1. It uses the subroutine CPROBE (CoarsePROBE) which is depicted in Algorithm 2. CPROBE is similar in spirit to the PROBE algorithm mentioned in the introduction and hence carries a similar name.

Representation of the Coarse Version. Algorithm 1 operates on an array $Y = (Y_{i1}, Y_{i2})_i$ of length $s = 4\lceil \frac{p}{\epsilon} \rceil$ consisting of tuples of integers. Throughout the algorithm, Y will represent

a $(S\epsilon/p)$-coarse version of the already seen prefix of the input integer sequence X, where S is the total weight of the already seen prefix. The coarse version can be explicitly computed from Y as in Line 10 of the algorithm: Replace every tuple (Y_{i1}, Y_{i2}) by the sum $Y_{i1} + Y_{i2}$.

The first element Y_{i1} of every non-zero tuple (Y_{i1}, Y_{i2}) will always equal a value $X[j]$, for some j, and Y_{i2} will always equal $\sum_{l=j+1}^{j'} X[l]$, for some $j' \geq j$. Such a tuple corresponds to the merging of the elements $X[j], \ldots, X[j']$ into a single element in the coarse version. We store the first element $X[j]$ and the remaining elements $\sum_{l=j+1}^{j'} X[l]$ separately as Y_{i1} and Y_{i2} in order to be able to guarantee that the crucial Item 3 of Definition 2 of a c-coarse version is fulfilled, namely, $Y_{i2} \leq c$, for some c.

Description of the Algorithm. In the first iteration of the while-loop, Y is filled with the first s integers of the input stream so that $Y = (X_1, 0), (X_2, 0), \ldots, (X_s, 0)$. CPROBE is then invoked on Y and parameter $\lfloor S\epsilon/p \rfloor$, where S is the current total weight of Y, i.e., $S = \sum_{i=1}^{s} Y_{i1} + Y_{i2}$ (or, equivalently, the weight of the prefix of the input stream seen so far). CPROBE(Y, c) computes an array of tuples $Z = (Z_{i1}, Z_{i2})_i$ representing a c-coarse version of Y. It greedily merges adjacent tuples $(Y_{i1}, Y_{i2}), \ldots, (Y_{j1}, Y_{j2})$ into a tuple $(Z_{k1}, Z_{k2}) = (Y_{i1}, Y_{i2} + \sum_{l=i+1}^{j} Y_{l1} + Y_{l2})$ for some k, where j is the largest value such that Z_{k2} does not exceed c. Note that the first parameter Z_{k1} takes the value of Y_{i1} unaltered. This guarantees that, throughout the algorithm, the first parameter of any non-zero tuple (Y_{i1}, Y_{i2}) always equals a value of the input stream $X[j]$, for some j.

We will prove in Lemma 5 that the length of the output sequence Z of CPROBE is at most $3p/\epsilon + 1$. Since Y is of length $4\lceil p/\epsilon \rceil$, in the next iteration of the while-loop of Algorithm 1, at least $p/\epsilon - 1$ new elements from the input stream are added to Y, which guarantees progress in every iteration of the while-loop. The process continues until the entire input stream has been processed.

Analysis. In the following, we will denote the input stream by $X \in \{1, \ldots, m\}^n$. For simplicity, we assume that $X[i] \gneq 0$, for all i, which is not a restriction since 0s in the input stream could simply be skipped by the algorithm.

▶ **Lemma 5.** *Consider the state of variable Y of Algorithm 1 at the beginning of iteration w of the while-loop, for any $w \geq 1$. Suppose that the prefix $X[1, q]$ of the input stream has been processed up until this point. Furthermore, let k be the largest j such that $Y[j] \neq (0, 0)$, and let $S = \sum_i Y_{i1} + Y_{i2}$. Then:*

1. $k \leq \frac{3p}{\epsilon} + 1$,
2. $Y_{i2} \leq S\epsilon/p$ *for every* $1 \leq i \leq s$,
3. *If $w \geq 2$, then there are integers $1 = t_1 < t_2 < \cdots < t_k < t_{k+1} = q + 1$ so that*

$$Y[1, k] = \left(X[t_1], \sum_{l=t_1+1}^{t_2-1} X[l]\right), \left(X[t_2], \sum_{l=t_2+1}^{t_3-1} X[l]\right), \ldots, \left(X[t_k], \sum_{l=t_k+1}^{t_{k+1}-1} X[l]\right).$$

Proof. We prove the statement by induction. Consider the first iteration. Then, all tuples of Y are $(0, 0)$ and $X[1, q]$ is an empty sequence, and the lemma is trivially true.

Now, suppose that the lemma is true in iteration $w \geq 1$. We will prove that it also holds in iteration $w + 1$.

Let $q' = q + (s - k - 1)$. Then, $Y[k + 1, s] = X[q + 1, q']$ after Line 6 of the algorithm (if $q' > n$ then suppose that a sequence of 0s follows the input stream). Let S' be the total weight of Y after the execution of Line 6 of the algorithm. Clearly, Item 3 is still fulfilled after appending elements of X to Y in Line 6. The left-to-right processing of Y

in CProbe guarantees that only consecutive elements are merged and that the order of Y stays intact. Furthermore, the first parameters of the tuples are copied (in Lines 3 and 8) and are therefore never changed, proving Item 3. The if-statement in Line 5 guarantees that merged elements do not exceed the value of $c = \lfloor S'\epsilon/p \rfloor$, thus ensuring Item 2. Let k' be the largest index such that $Y[k'] \neq (0,0)$, after the run of CProbe. To prove Item 1, notice that after the run of CProbe, for any $(Y_{i1}, Y_{i2}), (Y_{i+1,1}, Y_{i+1,2})$, we have $Y_{i2} + Y_{i+1,1} + Y_{i+1,2} > \lfloor S'\epsilon/p \rfloor$, since otherwise $(Y_{i+1,1}, Y_{i+1,2})$ would have been added to Y_{i2} in the algorithm. Thus, $S' = \sum_{i=1}^{k'} Y_{i1} + Y_{i2} > \frac{k'-1}{2} \cdot \lfloor S'\epsilon/p \rfloor$, which implies

$$k' < 1 + \frac{2S'p}{S'\epsilon - p} = 1 + \frac{2p}{\epsilon} + \frac{2p^2}{\epsilon(S'\epsilon - p)} \leq 1 + \frac{2p}{\epsilon} + \frac{2p^2}{\epsilon(\lceil 4\frac{p}{\epsilon} \rceil \epsilon - p)} < 1 + \frac{3p}{\epsilon},$$

where we used $S' \geq 4\lceil \frac{p}{\epsilon} \rceil$, since this quantity equals the length of Y, and all first elements of the tuples of Y are at least 1. ◀

▶ **Lemma 6.** *Algorithm 1 is a deterministic one-pass streaming algorithm that computes a $(S\epsilon/p)$-coarse version of the input stream $X \in \{1, \ldots, m\}^n$ using space $O(p \log(mn)/\epsilon)$, where $S = \sum_i X[i]$. It can be implemented with worst case update time $O(1)$.*

Proof. The structural properties (2) and (3) of Lemma 5 guarantee that the returned integer sequence Y' is a $(S\epsilon/p)$-coarse version of the input stream X and thus establish correctness of the algorithm. Property 1 ensures that the algorithm makes progress in every iteration: Since at most $\frac{3p}{\epsilon} + 1$ tuples of Y are different from $(0,0)$, in every iteration at least $s - (\frac{3p}{\epsilon} + 1) \geq \frac{p}{\epsilon} - 1$ integers from the input sequence are consumed.

Note that the algorithm as it is implemented in Algorithm 1 has amortized update time $O(1)$. Indeed, in every iteration of the while-loop, $\Theta(\frac{p}{\epsilon})$ integers from the input stream are consumed, and the runtime of one iteration is $O(\frac{p}{\epsilon})$. Using the following standard trick, we go from amortized update time to worst case update time: While executing the algorithm, we simultaneously read integers from the input stream into a buffer of size $\Theta(s)$ one at a time every $O(1)$ operations. Then, instead of consuming integers from the input stream in Line 5 of the algorithm, we consume the integers from the buffer. ◀

Using Lemma 6, we compute a $(S\epsilon/p)$-coarse version of the input stream, and we split it optimally in a post-processing step. This yields our main result of this section.

▶ **Theorem 7.** *There is a deterministic one-pass $(1+\epsilon)$-approximation streaming algorithm for Part with space $O(p \log(mn)/\epsilon)$, worst case update time $O(1)$, and $O(p/\epsilon)$ post-processing time.*

Proof. First, we will use Algorithm 1 in order to compute a coarse version of the input stream X. Let Y be the generated output which, according to Lemma 6, is a $(S\epsilon/p)$-coarse version of X. Then, we partition Y optimally using the exact linear time algorithm of Frederickson [9]. Following Lemma 3, we compute a $(1+\epsilon)$-approximation to an optimal partitioning of X.

Note that in order to deduce a partitioning of X from a partitioning of Y, it is required that every $Y[j]$ know the value $\min f^{-1}(j)$, where f is the coarsening function. This can be ensured by annotating the elements of the stream $X[i]$ by its position in the stream i, and forwarding those annotations to Y. ◀

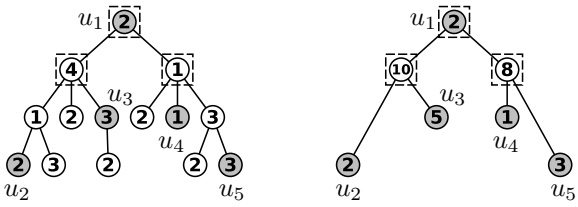

Figure 2 *Left*: An example tree. The highlighted nodes constitute set U of Definition 8. Set L of Definition 8 consists of the nodes within boxes. Note that the root node u_1 is by definition in B, however, it is also in L since $\mathrm{lca}_t(u_3, u_4) = u_1$. *Right*: The structure tree $\mathrm{ST}(U)$.

4 Algorithm for Partitioning Trees

We now present our algorithm for Tree. Our algorithm computes a *coarse structure tree*, i.e., a tree with much fewer nodes than the input tree t that captures the structure of t well enough so that an optimal partitioning of the structure tree allows us to obtain a $(1+\epsilon)$-approximation for Tree on t. In Subsec. 4.1, we define structure trees and we prove that an optimal partitioning of an appropriate structure tree approximates an optimal partitioning of t within a factor of $1+\epsilon$. This result is the most technical contribution of this paper. Then, in Subsec. 4.2, we argue that an appropriate structure tree can be computed with $\mathrm{O}(\frac{p^2}{\epsilon}\log(mn))$ space in the streaming model.

4.1 Structure Trees

Let t denote the weighted input tree that is described by the input stream X. Let the weight function of t be ω so that $\omega(v)$ denotes the weight of node $v \in t$, and let $\omega(t') := \sum_{v \in t'} \omega(v)$, for a subtree $t' \subseteq t$. Our definition of structure trees is as follows:

▶ **Definition 8** (Structure Tree). Let $U = \{u_1, \ldots, u_k\} \subseteq t$ be a set of *breakpoints* (nodes) ordered with respect to a depth-first-traversal of t, and suppose that $\mathrm{rt}(t) \in U$ (which implies that $u_1 = \mathrm{rt}(t)$). Let $L = \{\mathrm{lca}_t(u_i, u_{i+1}) \mid 1 \leq i \leq k-1\}$. Then the *structure tree* $\mathrm{ST}(U)$ of t with respect to U is a weighted tree with weight function ω' on vertex set $U \cup L$ such that there is an edge between $x, y \in U \cup L$ iff among the nodes $U \cup L$, y is the lowest ancestor of x in t. For a node $x \in U \cup L$, let $D(x) \subseteq t$ denote the set of nodes such that x is the lowest ancestor among the nodes $U \cup L$. Then, we define $\omega'(x) = \sum_{y \in D(x)} \omega(y)$.

Definition 8 is illustrated in Figure 2. An immediate consequence of the definition of a structure tree is that for every node $v \in \mathrm{ST}(U)$, the weight of the subtree rooted at v in $\mathrm{ST}(U)$ is the same as the weight of the subtree rooted at v in t. Consider a partitioning S of $\mathrm{ST}(U)$ (i.e., the roots of the subtrees induced by the partitioning). Since the weight of $\mathrm{stree}_{\mathrm{ST}(U)}(v)$ of any node $v \in \mathrm{ST}(U)$ is the same in tree t, clearly the bottleneck value of a partitioning S of $\mathrm{ST}(U)$ equals the bottleneck value of the partitioning S applied on t. This observation is similar to Item 1 of Lemma 3 for sequences, and is summarized in the following fact.

▶ **Fact 2.** *Let S be a partitioning of $\mathrm{ST}(U)$ with bottleneck value B. Then, S is also a partitioning of t with bottleneck value B.*

Choosing Good Breakpoints. The set of breakpoints U on which the structure tree $\mathrm{ST}(U)$ is built determines how well $\mathrm{ST}(U)$ represents the input tree t and thus how well a partitioning

of $\mathrm{ST}(U)$ approximates a partitioning of t. We will choose set U according to the following definition, which establishes a bridge between trees and integer sequences:

▶ **Definition 9** (c-coarse Structure Tree)**.** Let $V = \{v_1, \ldots, v_n\}$ denote the nodes of the input tree t ordered with respect to a depth-first-traversal of t. Let $X'[i] = \omega(v_i)$. For $U \subseteq V$ we say that $\mathrm{ST}(U)$ is a *c-coarse structure tree* of t if there exists a c-coarse version Y of X' of length $n' \leq n$ and coarsening function f such that $v_i \in U$ iff $\exists 1 \leq i' \leq n' : \min f^{-1}(i') = i$.

Consider the example tree in Figure 2. Then, we have $X' = 2\,4\,1\,2\,3\,2\,3\,2\,1\,2\,1\,3\,2\,3$. A 5-coarse version of X' is $7\,7\,6\,7\,3$ where X' has been grouped as follows: $\mathbf{2}\,4\,1\,|\,\mathbf{2}\,3\,2\,|\,\mathbf{3}\,2\,1\,2\,|\,\mathbf{1}\,3\,2\,|\,\mathbf{3}$. The numbers in bold are the first nodes of each block, and they correspond to the weights of those nodes of t that are put into set U. The structure tree in Figure 2 is built on this set U and is therefore a 5-coarse structure tree.

In the case of integer sequences, we showed that a coarse version of the input stream of size $\mathrm{O}(p/\epsilon)$ is sufficient to obtain a $(1+\epsilon)$-approximation. We will see that the more complicated structure of trees requires a coarse version of size $\Theta(p^2/\epsilon)$ for a $(1+\epsilon)$-approximation. Intuitively, this is due to the fact that, in the case of integer sequences, a partition is only adjacent to two other partitions. In the case of trees, a partition may be adjacent to $p-1$ other partitions. Hence, partition boundaries must be chosen more carefully, and a better resolution for our coarse object is required.

The main result of this Subsec., Lemma 10, states that, given a c-coarse structure tree $\mathrm{ST}(U)$, an optimal partitioning of $\mathrm{ST}(U)$ provides a partitioning of the original tree t such that the size of a partition increases at most by the additive term $2(p-1)c$. This is similar to Item 2 of Lemma 3 for integer sequences.

▶ **Lemma 10.** *Let U be such that $\mathrm{ST}(U)$ is a c-coarse structure tree of t. Consider an optimal partitioning S' of $\mathrm{ST}(U)$ into p parts. Consider the partitioning of t induced by S' and let B' denote the bottleneck value. Furthermore, let B^* denote the bottleneck value of an optimal partitioning of t. Then: $B' \leq B^* + 2(p-1)c$.*

Proof. Let S^* denote an optimal partitioning of t with bottleneck value B^*, let $t_1, \ldots, t_p$ denote the subtrees produced by S^*, and suppose that S^* is such that none of the subtrees t_i are empty (it is easy to see that there always is such a partitioning). Then, $S^* = \{\mathrm{rt}(t_1), \ldots, \mathrm{rt}(t_p)\}$. Denote by $U_i \subseteq U$ the subset of nodes of U that are also contained in t_i. Given S^*, we construct a partitioning $\tilde{S} = \{\tilde{s}_1, \ldots, \tilde{s}_p\}$ of $\mathrm{ST}(B)$ with bottleneck value $\tilde{B} \leq B^* + 2(p-1)c$. Since the optimal partitioning S' of $\mathrm{ST}(B)$ is at least as good, Fact 2 then implies the result.

Definition of $\tilde{S}$. For each partition t_i, we define a vertex $\tilde{s}_i$ of the partitioning $\tilde{S}$. Ideally, for every i, we would like to set $\tilde{s}_i = \mathrm{rt}(t_i)$, however, we can only do that if $\mathrm{rt}(t_i) \in \mathrm{ST}(U)$. If this is not the case, we select a nearby node contained in t_i that is also included in $ST(U)$, or, if the weight of t_i is not significant enough, we do not select any node, giving an empty partition. The definition of $\tilde{s}_i$ is as follows:

1. If $\mathrm{rt}(t_i) \in \mathrm{ST}(U)$: Let $\tilde{s}_i = \mathrm{rt}(t_i)$.
2. If $\mathrm{rt}(t_i) \notin \mathrm{ST}(U)$ and $\omega(t_i) \leq 2c$: Let $\tilde{s}_i = \perp$ (indicating that $\tilde{s}_i$ is unused).
3. If $\mathrm{rt}(t_i) \notin \mathrm{ST}(U)$ and $w(t_i) > 2c$: We define $\tilde{s}_i = \mathrm{lca}_t(B_i)$.
   For this to be a valid assignment, we will show in Lemma 12 that B_i is non-empty and in Lemma 13 that $\mathrm{lca}_t(B_i) \in \mathrm{ST}(B)$. We will prove in Lemma 14 that $\tilde{s}_i = \mathrm{lca}_t(B_i)$ is in a sense close to $\mathrm{rt}(t_i)$ by showing the following inequality on which we base our analysis: $\omega(\mathrm{stree}_t(\tilde{s}_i)) \geq \omega(\mathrm{stree}_t(\mathrm{rt}(t_i))) - 2c$.

Bounding $\tilde{B}$. We will now prove that the bottleneck value $\tilde{B}$ of the partitioning given by $\tilde{S}$ is at most $B^* + 2(p-1)c$. To see this, let $\tilde{t}_1, \ldots, \tilde{t}_p$ denote the subtrees induced by the partitioning $\tilde{S}$ such that $\mathrm{rt}(\tilde{t}_i) = \tilde{s}_i$ (if $\tilde{s}_i = \perp$ then let $\tilde{t}_i = \perp$). We will prove that $w(\tilde{t}_i) \leq \omega(t_i) + 2(p-1)c$, for every $\tilde{t}_i \neq \perp$, which then implies the result.

Now consider one partition $\tilde{t}_i$ with $\tilde{t}_i \neq \perp$. Let

$$J = \{j : \mathrm{rt}(t_j) \text{ is connected to a leaf of } t_i \text{ in } t\},$$

and let $J_\perp \subseteq J$ be those indices j such that $\tilde{s}_j = \perp$. Then:

$$\begin{aligned}
\omega(t_i) &= \omega(\mathrm{stree}_t(\mathrm{rt}(t_i))) - \sum_{j \in J} \omega(\mathrm{stree}_t(\mathrm{rt}(t_j))) \\
&\geq \omega(\mathrm{stree}_t(\tilde{s}_i))) - \sum_{j \in J_\perp} \omega(\mathrm{stree}_t(\mathrm{rt}(t_j))) - \sum_{j \in J \setminus J_\perp} \omega(\mathrm{stree}_t(\mathrm{rt}(t_j))), \qquad (1)
\end{aligned}$$

where we used $\omega(\mathrm{stree}_t(\mathrm{rt}(t_i))) \geq \omega(\mathrm{stree}_t(\tilde{s}_i)))$ since $\tilde{s}_i$ is contained in t_i. Then, due to Item 2 of the previous case distinction, we have $\omega(\mathrm{stree}_t(\mathrm{rt}(t_j))) \leq 2c$ for every $j \in J_\perp$, and, for every $j \in J \setminus J_\perp$, according to Items 1 and 3 of our case distinction, we have $\mathrm{stree}_t(\mathrm{rt}(t_j)) - \mathrm{stree}_t(\tilde{s}_j) \leq 2c$. Thus:

$$\begin{aligned}
\omega(t_i) &\geq \ldots \geq \omega(\mathrm{stree}_t(\tilde{s}_i))) - |J_\perp|2c - \sum_{j \in J \setminus J_\perp} (\omega(\mathrm{stree}_t(\tilde{s}_j)) + 2c) \\
&= \omega(\mathrm{stree}_t(\tilde{s}_i))) - \left(\sum_{j \in J \setminus J_\perp} \omega(\mathrm{stree}_t(\tilde{s}_j)) \right) - |J_\perp|2c - |J \setminus J_\perp|2c \\
&\geq \omega(\tilde{t}_i) - |J|2c \geq \omega(\tilde{t}_i) - (p-1)2c,
\end{aligned}$$

where we used $\omega(\tilde{t}_i) \leq \omega(\mathrm{stree}_t(\tilde{s}_i))) - \left(\sum_{j \in J \setminus J_\perp} \omega(\mathrm{stree}_t(\tilde{s}_j))\right)$ and $|J| \leq p-1$. The result follows. ◀

▶ **Corollary 11.** *Let U be such that $\mathrm{ST}(U)$ is a $S\epsilon/(2p^2)$-coarse structure tree where $S = \omega(t)$ for some $\epsilon > 0$. Then, an optimal partitioning of $\mathrm{ST}(U)$ into p parts provides a $(1+\epsilon)$-approximation to* Tree *on t.*

Proof. Let B' be the bottleneck value of an optimal partitioning P' of $\mathrm{ST}(U)$, and let B^* be the bottleneck value of an optimal partitioning of t. Then, by Fact 2, P' induces a partitioning of t with bottleneck value B'. Next, by Lemma 10, we have $B' \leq B^* + 2(p-1)c$, where $c = \epsilon/(2p^2)$. Furthermore, notice that $B^* \geq S/p$. Thus, we obtain the approximation ratio: $\frac{B'}{B} \leq \frac{B^* + 2(p-1)\cdot S\epsilon/(2p^2)}{B^*} < 1 + \frac{S\epsilon}{pB^*} \leq 1 + \epsilon$. ◀

Auxiliary Lemmas Used in the Proof of Lemma 10. We now present the technical lemmas that have been used in the proof of Lemma 10.

In Lemma 12, we show that for every subtree $\mathrm{stree}_t(v)$ of weight at least c, for some node v, at least one node of $\mathrm{stree}_t(v)$ is contained in every c-coarse structure tree.

▶ **Lemma 12 (*).** *Let U be such that $\mathrm{ST}(U)$ is a c-coarse structure tree of t. Let $v \in t$ be any node so that $\omega(\mathrm{stree}_t(v)) > c$. Then, $U \cap \mathrm{stree}_t(v) \neq \varnothing$.*

Lemma 13 is a simple structural property of trees.

▶ **Lemma 13.** *Let $U = \{u_1, u_2 \ldots\} \subseteq t$ be a subset of nodes of a tree of size at least two, ordered with respect to a depth-first-traversal. Then, there is an index $1 \leq i \leq |U| - 1$ such that $\mathrm{lca}(u_i, u_{i+1}) = \mathrm{lca}(U)$.*

Proof. Let $U_i = \{u_1, \dots, u_i\}$. We prove by induction on i that the statement is true for all sets U_i. Suppose that $i = 2$. Then trivially $\mathrm{lca}(u_1, u_2) = \mathrm{lca}(U_2)$. Now suppose that the statement is true for i and let x denote the $\mathrm{lca}(u_j, u_{j+1})$ for the value of j with $1 \leq j \leq i-1$ so that $x = \mathrm{lca}(U_i)$. If $u_{i+1} \in \mathrm{stree}_t(x)$ then clearly x is also the lowest-common-ancestor of U_{i+1}. Suppose now that $u_{i+1} \notin \mathrm{stree}_t(x)$. Let $y = \mathrm{lca}(u_i, u_{i+1})$. Then clearly $x \in \mathrm{stree}_t(y)$ and hence $U_i \subseteq \mathrm{stree}_t(y)$. Therefore, y is an ancestor of every node of U_{i+1}. It is also the lowest-common-ancestor of all nodes U_{i+1} as it is defined as $\mathrm{lca}(u_i, u_{i+1})$. ◀

Given a subtree $\mathrm{stree}_t(v)$ of weight at least c, for some node v, we show in Lemma 14 that the lowest-common-ancestor x of the nodes $U' = U \cap \mathrm{stree}_t(v)$ of a c-coarse structure tree $\mathrm{ST}(U)$ is in a sense close to v, i.e., $\omega(\mathrm{stree}_t(x)) + 2c \geq \omega(\mathrm{stree}_t(v))$.

▶ **Lemma 14 (*).** *Let U be so that $\mathrm{ST}(U)$ is a c-coarse structure tree of t. Let $v \in t$ be any node such that $\omega(\mathrm{stree}_t(v)) > c$, and let $U' = U \cap \mathrm{stree}_t(v)$. Furthermore, let $x = \mathrm{lca}(U')$. Then: $\omega(\mathrm{stree}_t(x)) + 2c \geq \omega(\mathrm{stree}_t(v))$.*

4.2 Computing Structure Trees

Algorithm. We use the first and the second pass to compute the IDs of the nodes $B \cup L$ with $B = \{b_1, \dots, b_k\}$ (we assume that they are ordered w.r.t. a depth-first-traversal of t) so that $\mathrm{ST}(B)$ is a $S\epsilon/(2p^2)$-coarse structure tree of t and $L = \{\mathrm{lca}(b_i, b_{i+1}) \,|\, 1 \leq i \leq k-1\}$. Then, in the third pass, we establish $\mathrm{ST}(B)$.

1st pass. Consider the subsequence of down-steps X_d of the input stream X and let X' denote the sequence of weights that constitute the second parameters of the tuples of X_d. Compute a $(\frac{S\epsilon}{2p^2})$-coarse version of X' by running the algorithm for partitioning integer sequences on X'[4]. Annotate every breakpoint of B created by the algorithm with the ID of the node that lead to its creation. This guarantees access to the IDs of the nodes B.

2nd pass. Concerning the nodes in L, we make use of the following observation: Let b_i, b_{i+1} be two nodes for which breakpoints are stored and $b_{i+1} \notin \mathrm{stree}_t(b_i)$. Let d_i be the minimal depth of a down-step between the down-step describing b_i and the down-step describing b_{i+1}. Then $x = \mathrm{lca}_t(b_i, b_{i+1})$ has depth $d_i - 1$. In the first pass, while simultaneously running the algorithm for partitioning integer sequences, we compute this depth (by keeping track of the minimal depth between any two nodes stored as first elements of two consecutive tuples in variable Y in Algorithm 1). Then, the down-step of node x, and hence the ID of x, can be identified in a second pass as the down-step at depth $d_i - 1$ that appears before the down-step of b_i, but is closest to the down-step b_i in the input stream.

3rd pass. Let $I = \{i_1, i_2, \dots\}$ be the IDs of the nodes $B \cup L$ and suppose that $i_j \leq i_{j+1}$ for every $1 \leq j < |I|$. We describe how to build the structure tree inductively. Suppose that it is correctly built up to the node with ID i_j, that is, it is the correct structure tree of the subtree of t induced by all nodes with ID at most i_j. The substream $X_j = (\text{'d'}, w_j), \dots, (\text{'d'}, w_{j+1})$, where w_j and w_{j+1} are the weights of the nodes $\mathrm{node}_t(i_j)$ and $\mathrm{node}_t(i_{j+1})$, respectively, allows us to determine the relationship between $\mathrm{node}_t(i_j)$ and $\mathrm{node}_t(i_{j+1})$. If $\mathrm{node}_t(i_{j+1})$

[4] Algorithm 1 computes a $(\frac{S\epsilon}{p})$-coarse version. In order to compute a $(\frac{S\epsilon}{2p^2})$-coarse version instead, CProbe has to be invoked with parameter $\lfloor \frac{S\epsilon}{2p^2} \rfloor$ in Line 8, and s should be set to $8\lceil \frac{p^2}{\epsilon} \rceil$ in Line 1.

is in the subtree of $\text{node}_t(i_j)$, then we add an edge from $\text{node}_t(i_j)$ to $\text{node}_t(i_{j+1})$ in $\text{ST}(B)$. Otherwise, we consider the minimum depth d_j of a down-step in the substream X_j. We deduce that $x = \text{lca}_t(\text{node}_t(i_j), \text{node}_t(i_{j+1}))$ is at depth $d_j - 1$. Node x has already been added to $\text{ST}(B)$, and, by its depth, we can identify it and connect it to $\text{node}_t(i_{j+1})$ in $\text{ST}(B)$. Concerning updating the weights ω', for every down-step $(\text{'d'}, w)$ in the stream, we add the weight w to the closest ancestor c in the current $\text{ST}(B)$ of the node that is described by the current down-step.

Post-processing. We can therefore establish $\text{ST}(B)$ in three passes. As a post-processing step, we use an optimal linear time partitioning algorithm by Frederickson [9] in order to partition $\text{ST}(B)$. From the resulting partitioning, according to Corollary 11, we deduce a $(1+\epsilon)$-approximation to the partitioning of t. All steps other from running the algorithm of Theorem 7 can be implemented with $\text{O}(1)$ update time.

Furthermore, it can be seen that the second and the third passes can be merged into a single pass (not explicitly described here), leading to a two-pass algorithm.

▶ **Theorem 15.** *There is a det. two-pass $(1+\epsilon)$-approximation streaming algorithm for* Tree *with space $\text{O}(\frac{p^2}{\epsilon}\log(mn))$, worst case update time $\text{O}(1)$, and post-processing time $\text{O}(p^2/\epsilon)$.*

5 Lower Bounds for Partitioning Integer Sequences

5.1 A Linear Space Lower Bound for Exact Algorithms

In this section, we show that any possibly randomized exact streaming algorithm for Part that performs one pass over the input requires $\Omega(n)$ space. We show this by a reduction from the Index problem in one-way two-party communication complexity.

▶ **Definition 16** (Index Problem)**.** Let $S = (S_1, \ldots, S_N)$ where $S \in \{0,1\}^N$, and let $I \in \{1, \ldots, N\}$. Alice is given S, Bob is given I. Alice sends message M to Bob and, upon reception, Bob outputs S_I.

We consider a version of Index where the index I is chosen from the set $\{\lceil N/2\rceil, \ldots, N\}$ uniformly at random. It is well-known [17] that the one-way randomized communication complexity of Index is $\Omega(N)$, and the modification in the input distribution restricting the index I to be chosen from the set $\{\lceil N/2\rceil, \ldots, N\}$ clearly changes the communication complexity only by a constant factor.

▶ **Lemma 17** (Hardness of the Index Problem)**.** *If S is chosen uniformly at random from $\{0,1\}^N$, I is chosen uniformly at random from the set $\{\lceil N/2\rceil, \ldots, N\}$, and the failure probability of the protocol is at most $1/3$, then $\mathbb{E}_S|M| = \Omega(N)$.*

Reduction. Given a streaming algorithm ALG that solves Part on a stream of length at most $3n$ using space s, we specify a protocol for an arbitrary instance (S, I) of Index with $|S| = n$, such that the message size is at most s.

Remember that Alice holds $S \in \{0,1\}^N$ and Bob holds $I \geq \lceil N/2\rceil$. Our protocol is the following: Alice generates the sequence $Y \in \{1,3\}^{2N}$ such that $Y_i = 2 \cdot S_{i/2} + 1$ for even i, and $Y_i = 4 - Y_{i+1}$ for odd i. Bob generates the sequence $Z = \underbrace{4 \ldots 4}_{2I-N-1}\ 2$.

Alice runs ALG on sequence Y with number of partitions $p = 2$. Once Y is entirely processed, she sends the resulting memory state of ALG to Bob. Bob continues running

ALG on Alice's final memory state and feeds the sequence Z into ALG. The message size of the protocol equals the space usage of ALG after processing Y. Note that ALG outputs the separator s_1 that separates the two partitions. If s_1 is even then Bob outputs 0, and if s_1 is odd then Bob outputs 1.

We prove that the above protocol is correct, which immediately yields the result.

▶ **Theorem 18.** *Any possibly randomized one-pass streaming algorithm for* Part *with error probability at most* $1/3$ *requires space* $\Omega(n)$.

Proof. First observe that $\sum_i Y_i + \sum_i Z_i = 4 \cdot N + (2I - N - 1) \cdot 4 + 2 = 8I - 2$. Let s_1^* denote the optimal split position. Suppose that a perfect balancing is achieved and the optimal bottleneck value is $4I - 1$. Since for all $i = 1, \ldots, N$ we have $Y_{2i-1} + Y_{2i} = 4$, this can only be achieved if s_1^* is even and $Y_{s_1^*} = 1$ which implies that $S_{s_1^*/2} = S_I = 1$. Suppose now that a perfect balancing cannot be achieved. This can only happen if s_1^* is odd and $Y_{s_1^*-1} = 3$ which implies that $S_{(s_1^*-1)/2} = S_I = 3$. Thus, the protocol is correct in both cases, and ALG can be used to solve Index. Lemma 17 implies the result. ◀

5.2 $\Omega(\frac{1}{\epsilon} \log n)$ Space Lower Bound for Approximation Algorithms

We prove now an $\Omega(\frac{1}{\epsilon} \log n)$ space lower bound for one-pass algorithms for Part that compute a $(1+\epsilon)$-approximation. We prove this lower bound in the one-way two-party communication setting for instances of Part with $m = 1$ and $p = 2$. Alice is given a sequence $Y \in \{0,1\}^n$ and Bob is given a sequence $Z \in \{0,1\}^n$, and they have to split the sequence $X = Y \circ Z$ into two parts. Alice sends a message to Bob, and, upon reception, Bob outputs the separator.

Input Distribution. Let t be an integer that is to be determined later. Alice's input and Bob's input are independent from each other and they are constructed as follows:

- **Alice's input** Y is a sequence of length n with $2(t-1)$ leading 1s, followed by an arbitrary sequence of length $n - 3t + 2$ with elements from $\{0, 11\}$ (11 is a pair of ones), where the number of 11s is exactly t. Denote by $\mathcal{Y}$ the set of all such sequences. Then Y is chosen uniformly at random from $\mathcal{Y}$. Clearly, the weight of Y is $4t - 2$, and $|\mathcal{Y}| = \binom{n-3t+2}{t}$.
- **Bob's input** Z is a sequence of length n with the first $4(i-1)$ elements 1, and the remaining elements 0, for some $i \in \{1, 2, \ldots, t\}$. Denote all such sequences as $\mathcal{Z}$. Then Z is chosen uniformly at random from $\mathcal{Z}$. Observe that the weight of Z varies from 0 to $4(t-1)$, and $|\mathcal{Z}| = t$.

Note that an optimal partitioning of any $Y \circ Z$ instance splits one of the 11s in the second part of Alice's input.

Example: Let $t = 2$, $n = 10$, and $p = 2$. Suppose that Alice holds $Y = 11\,00110110$. Bob's possible inputs are $Z_1 = 0000000000$ and $Z_2 = 1111000000$ of weight 0 and 4, respectively. The optimal partitioning of $Y \circ Z_1$ is $11\,001 \,|\, 10110\,0 \ldots 0$ and of $Y \circ Z_2$ is $11\,001101 \,|\, 10\,11110 \ldots 0$.

We give a lower bound on the communication complexity of any possibly randomized communication protocol that solves instances of $\mathcal{Y} \times \mathcal{Z}$ exactly.

▶ **Lemma 19.** *Any randomized one-way two-party protocol with error at most* $\delta > 0$ *that solves* Part *on instances of* $\mathcal{Y} \times \mathcal{Z}$ *has communication complexity at least* $\log\left(\frac{\binom{n-3t+2}{t}}{8\binom{t}{4\delta t} n^{4\delta t}}\right)$.

Proof. Let P be a randomized protocol as in the statement of the lemma. Then, by Yao's Lemma [28], there is a deterministic protocol Q with distributional error at most δ that has the same communication complexity. We prove a lower bound on the communication complexity of Q.

Denote by $M_1, \ldots, M_k$ the possible messages from Alice to Bob, and let $\mathcal{Y}_i \subseteq \mathcal{Y}$ denote the set of inputs that Alice maps to message M_i. Note that for a fixed input for Bob, the protocol Q outputs the same result for all inputs in $\mathcal{Y}_i$. Let $p_i = \Pr_{Y \leftarrow \mathcal{Y}_i, Z \leftarrow \mathcal{Z}}[Q \text{ errs on } (Y,Z)]$. Since the distributional error of the protocol is δ, or, in other words, $\Pr_{Y \leftarrow \mathcal{Y}, Z \leftarrow \mathcal{Z}}[Q \text{ errs on } (Y,Z)] \leq \delta$, we obtain $\frac{\sum_i p_i |\mathcal{Y}_i|}{|\mathcal{Y}|} \leq \delta$. Let $i \in \{1, \ldots, l\}$ be the indices for which $p_i \leq 2\delta$. Then by the Markov Inequality, $\sum_{i=1}^{l} |\mathcal{Y}_i| \geq \frac{1}{2}|\mathcal{Y}|$.

We bound $|\mathcal{Y}_i|$ from above for all $i \in \{1, \ldots, l\}$. First, note that for a particular input $Z \in \mathcal{Z}$, the output of Q on (Y,Z) is the same for all $Y \in \mathcal{Y}_i$. Denote by $\mathcal{Y}_i^j$ the subset of $\mathcal{Y}_i$ such that for each $Y^j \in \mathcal{Y}_i^j : \Pr_{Z \leftarrow \mathcal{Z}}[Q \text{ errs on } (Y^j, Z)] = \frac{j}{t}$, or, in other words, there are j inputs of Bob such that the protocol fails on Y^j, and, for the remaining $t-j$ inputs of Bob, the protocol succeeds. Consider the set $\mathcal{Y}_i^0$, i.e., for each $Y \in \mathcal{Y}_i^0$, the protocol succeeds on any input of Bob. This determines all positions of the pairs of 1s in Alice's input, and, therefore, there is only a single such element and we obtain $|\mathcal{Y}_i^0| \leq 1$. Similarly, we obtain $|\mathcal{Y}_i^j| \leq \binom{t}{j} n^j$, since the protocol errs on at most j inputs of Bob, therefore the position of $t-j$ pairs of 1s is fixed and only j pairs of 1s may differ (we allow them to have an arbitrary position in Y which is a very rough but sufficient estimate).

We apply the Markov Inequality again: For at least half of the elements of $\mathcal{Y}_i$, the protocol errs with probability at most 4δ. Therefore, $\frac{1}{2}|\mathcal{Y}_i| \leq \sum_{j \leq 4\delta t} |\mathcal{Y}_i^j| \leq \sum_{j \leq 4\delta t} \binom{t}{j} n^j \leq 2\binom{t}{4\delta t} n^{4\delta t}$, and thus $|\mathcal{Y}_i| \leq 4\binom{t}{4\delta t} n^{4\delta t}$. This then implies $l \geq \frac{|\mathcal{Y}|}{8\binom{t}{4\delta t} n^{4\delta t}} = \frac{\binom{n-3t+2}{t}}{8\binom{t}{4\delta t} n^{4\delta t}}$. Since the protocol sends at least l different messages, the communication complexity of the protocol is at least $\log(l)$, which implies the result. ◀

We make t small enough so that a solution to any instance of $\mathcal{Y} \times \mathcal{Z}$ that is a $(1+\epsilon)$-approximation actually solves the instance exactly. This idea leads to the following theorem:

▶ **Theorem 20.** *Any randomized one-way two-party communication protocol with error at most $\delta > 0$ (δ sufficiently small) that computes a $(1+\epsilon)$-approximation ($\frac{1}{\epsilon} = O(n^{1-\gamma})$ for any $\gamma > 0$) to* Part *on instances of $\mathcal{Y} \times \mathcal{Z}$ has communication complexity at least $\Omega\left(\frac{1}{\epsilon} \log n\right)$.*

Proof. We choose t small enough that a solution to any instance of $\mathcal{Y} \times \mathcal{Z}$ that is a $(1+\epsilon)$-approximation actually solves the instance exactly. Remark again that the weight of Y is $4t-2$ and the weight of Z is $4(i-1)$. Since the total weight is even, there is always a partitioning with weight $2t-1+2(i-1)$. Therefore, any partitioning that does not achieve an optimal balancing has an approximation factor of at least $\frac{2t-1+2(i-1)+1}{2t-1+2(i-1)}$, and we wish to choose t such that this approximation factor is worse than a $(1+\epsilon)$ approximation. Therefore, we have to choose t small enough such that for any $i \in \{1, 2, \ldots, t\}$: $\frac{1}{2t-1+2(i-1)} > \epsilon$, which implies that $t < \frac{1}{4\epsilon} + \frac{3}{4}$. We choose $t = \frac{1}{4\epsilon}$ and plug this value into the communication lower bound from Lemma 19. Using standard bounds on binomial coefficients:

$$\begin{aligned}
\Omega\left(\log\left(\frac{\binom{n-3t+2}{t}}{8\binom{t}{4\delta t} n^{4\delta t}}\right)\right) &= \Omega\left(\log\left(\frac{\left(4\epsilon(n-\frac{3}{4\epsilon}+2)\right)^{\frac{1}{4\epsilon}}}{8n^{\delta/\epsilon}\left(\frac{e}{4\delta}\right)^{\delta/\epsilon}}\right)\right) \\
&= \Omega\left(\frac{1}{4\epsilon}\log(4\epsilon n - 3 + 8\epsilon) - \frac{\delta}{\epsilon}\log(\frac{ne}{4\delta})\right) \\
&= \Omega\left(\frac{1}{4\epsilon}\log(4\epsilon n) - \frac{\delta}{\epsilon}\log(\frac{ne}{4\delta})\right) = \Omega(\left(\frac{1}{\epsilon}\log n\right),
\end{aligned}$$

for a sufficiently small but constant δ, and $\epsilon = \mathrm{O}(n^{1-\gamma})$ for any $\gamma > 0$. The result follows. ◀

6 Conclusion

In this paper, we initiated the study of the problems of partitioning integer sequences and partitioning trees in the streaming model. We showed that, for both problems, smaller versions of the input instances can be computed in a streaming fashion and still capture $(1+\epsilon)$-approximate partitionings of the original instances. For integer sequences, the small instances are of size $O(\frac{p}{\epsilon})$, and for trees, the small instances are of size $O(\frac{p^2}{\epsilon})$, both independent of the length of the input stream. Furthermore, for the problem of partitioning integer sequences, we provided space lower bounds obtained through communication complexity.

It remains to be investigated whether the sizes of the small instances for trees can be reduced to $O(\frac{p}{\epsilon})$. Furthermore, we conjecture that the number of passes of our algorithm for Tree can be reduced from two to one.

Acknowledgements. The author thanks László Kozma for many valuable ideas and helpful discussions, and an anonymous reviewer for improving the baseline algorithm stated in Fact 1.

References

1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 31st Symposium on Principles of Database Systems*, PODS'12, pages 5–14, New York, NY, USA, 2012. ACM.

2 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In Prasad Raghavendra, Sofya Raskhodnikova, Klaus Jansen, and JoséD.P. Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, volume 8096 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-40328-6_1.

3 Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'04, pages 120–124, New York, NY, USA, 2004. ACM. doi:10.1145/1007912.1007931.

4 Michael Bader. *Space-Filling Curves – An Introduction with Applications in Scientific Computing*. Texts in Computational Science and Engineering. Springer, 2013.

5 Joshua Batson, Daniel A. Spielman, Nikhil Srivastava, and Shang-Hua Teng. Spectral sparsification of graphs: Theory and algorithms. *Commun. ACM*, 56(8):87–94, August 2013.

6 S. H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. Comput.*, 37(1):48–57, January 1988. doi:10.1109/12.75137.

7 Vanessa Braganholo and Marta Mattoso. A survey on xml fragmentation. In *SIGMOD'14*, New York, NY, USA, 2014. ACM.

8 Stefan Fafianie and Stefan Kratsch. Streaming kernelization. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014*, volume 8635 of *Lecture Notes in Computer Science*, pages 275–286. Springer Berlin Heidelberg, 2014.

9 Greg N. Frederickson. Optimal algorithms for tree partitioning. In *SODA'91*, pages 168–177, Philadelphia, PA, USA, 1991. URL: http://dl.acm.org/citation.cfm?id=127787.127822.

10 Yijie Han, Bhagirath Narahari, and Hyeong-Ah Choi. Mapping a chain task to chained processors. *Inf. Process. Lett.*, 44(3):141–148, 1992. doi:10.1016/0020-0190(92)90054-Y.

11 Pierre Hansen and Keh-Wei Lih. Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. Comput.*, 1992.

12 Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 561–570, 2014.

13 Sanjeev Khanna, S. Muthukrishnan, and Steven Skiena. Efficient array partitioning. In *ICALP*, volume 1256, pages 616–626. Springer Berlin Heidelberg, 1997. doi:10.1007/3-540-63165-8_216.

14 Jin Kim and Hyunchul Kang. A method of XML document fragmentation for reducing time of XML fragment stream query processing. *Computing and Informatics*, 31(3):639, 2012. URL: http://www.cai.sk/ojs/index.php/cai/article/view/1012.

15 Christian Konrad. Two-constraint domain decomposition with space filling curves. *Parallel Comput.*, 37(4-5):203–216, April 2011. doi:10.1016/j.parco.2011.03.002.

16 Christian Konrad and Frédéric Magniez. Validating xml documents in the streaming model with external memory. *ACM Trans. Database Syst.*, 38(4), 2013. doi:10.1145/2504590.

17 Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997.

18 Daniel Lokshtanov, Neeldhara Misra, and Saket Saurabh. Kernelization – preprocessing with a guarantee. In *The Multivariate Algorithmic Revolution and Beyond – Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*, pages 129–161, 2012.

19 F. Magniez, C. Mathieu, and A. Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM Journal on Computing*, 2014.

20 Fredrik Manne and Bjørn Olstad. Efficient partitioning of sequences. *IEEE Trans. Comput.*, 44(11):1322–1326, November 1995. doi:10.1109/12.475128.

21 Fredrik Manne and Tor Sørevik. Optimal partitioning of sequences. *J. Algorithms*, 19(2):235–249, September 1995. doi:10.1006/jagm.1995.1035.

22 Serge Miguet and Jean-Marc Pierson. Heuristics for 1d rectilinear partitioning as a low cost and high quality answer to dynamic load balancing. In *HPCN Europe 1997*, pages 550–564, London, UK, UK, 1997. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645561.659355.

23 Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, FOCS'10, pages 815–823, Washington, DC, USA, 2010. IEEE Computer Society.

24 Yehoshua Perl and Stephen R. Schach. Max-min tree partitioning. *J. ACM*, 28(1):5–15, January 1981. doi:10.1145/322234.322236.

25 Ali Pinar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *J. Parallel Distrib. Comput.*, 64(8):974–996, August 2004. doi:10.1016/j.jpdc.2004.05.003.

26 Stefan Schamberger and Jens-Michael Wierum. Partitioning finite element meshes using space-filling curves. *Future Gener. Comput. Syst.*, 21(5):759–766, May 2005. doi:10.1016/j.future.2004.05.018.

27 Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC'01, pages 183–192, New York, NY, USA, 2001. ACM.

28 Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *FOCS'77*, pages 222–227, Washington, DC, USA, 1977. doi:10.1109/SFCS.1977.24.

29 Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *PLDI'13*, 2013. doi:10.1145/2491956.2462159.